Réalisation de nos premiers widgets

version 1

Interface Homme-Machine : Unity

Voici les objectifs de ce sujet :

— Continuer à manipuler l'IDE `Unity`.

— Création d'un *widget* complexe.

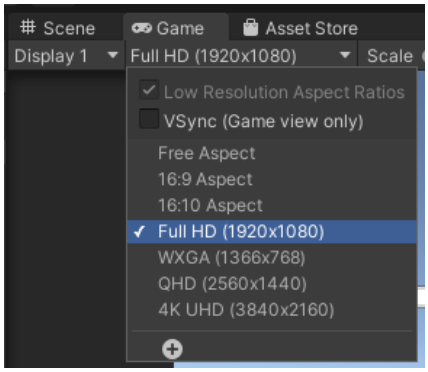
— Mécanisme des préfabriqués.

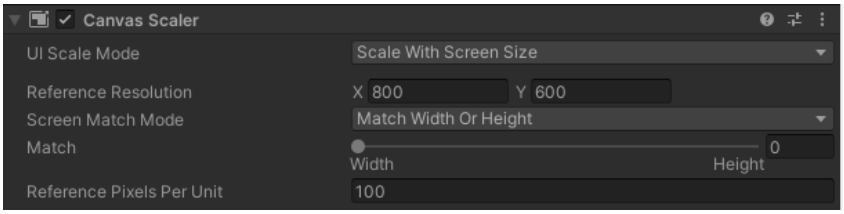
— Exportation de son travail : les *unity package*.

Description générale du TP

La fois précédente, nous avons réalisé notre premier projet Unity et nous avons vu la manipulation de l'interface. La manipulation est simple (mais nécessite quelques calibrations). Nous avons réalisé entre autres notre première interface très simplifiée, reposant sur la mécanique des ancres (voire aucune pour certains d'entre vous). La mécanique des ancres n'est pas aisée, surtout lorsque les ratios de l'écran peuvent changer (réalisation d'une application `PC` et une application `Android`). Ce dernier point ne sera pas étudié dans ce TP car cela rentre dans les aspects très avancés. Cependant, pour le lecteur volontaire ou si vous finissez le TP (donc pas au début) il serait intéressant de regarder ce pointeur `https://docs.unity3d.com/2020.1/Documentation/Manual/HOWTO-UIMultiResolution.html`.

Quoi qu'il arrive, on vous demande dans un premier temps de créer un **nouveau** projet vide et de fixer une résolution raisonnable [1] de votre `Canvas`, dans le menu `Game` de la fenêtre (sinon il s'agit de paramètre spécifique au déploiement de votre application). Modifier aussi en sélectionnant dans le `Canvas` de la hiérarchie, le composant gérant la mise à l'échelle en fonction de la résolution (`Canvas Scaler`) et respectant les paramètres (à la résolution près) de l'image de droite.

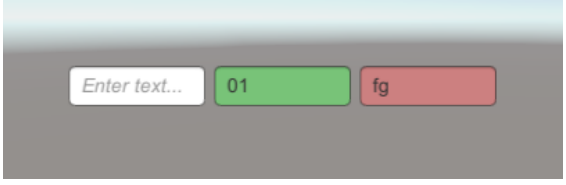




Une fois cela réalisé, nous pouvons passer à la réalisation d'un widget complexe. Pour expliquer ce terme, nous allons réaliser un agglomérat de widgets existants avec un ou plusieurs scripts pour régir le comportement global du widget.

1 Premier widget complexe : FormattedInputField

L'objectif de ce widget est de réaliser une zone de texte qui change de couleur selon une expression régulière particulière comme sur l'image ci-dessous, où nous avons 3 `FormattedInputField`. Ainsi, lorsque la saisie est vide (le widget de gauche), nous avons une couleur standard. La présence d'un nombre engendre une couleur (le widget au centre) et une autre couleur si l'expression régulière n'est pas présente (le widget à droite).



Pour cela, nous vous donnons le code suivant qui vient d'un programme `C#` basique en dehors de `Unity` :

1. La résolution que j'ai dans mes versions est par défaut le `1024x768`.

```
Console.WriteLine("Regex␣experimentation");
string regex = "[0-9]+";
string montext = "bonjour";

if (System.Text.RegularExpressions.Regex.IsMatch(montext, regex))
       Console.WriteLine("Le␣texte␣matche␣la␣regex");
else
       Console.WriteLine("le␣texte␣ne␣matche␣pas␣la␣regex");
```

D'un point de vue conceptuel, nous devons suivre les étapes suivantes (si vous faites autrement, tant pis pour vous, même si cela peut fonctionner) :

1. Créer les attributs publics correspondant aux couleurs et les initialiser directement.
2. Créer l'attribut de la `regex` sous forme de chaîne de caractères.
3. Réaliser la `callback` lorsqu'on change le texte dans le champ de saisie.
4. La couleur qu'il faut changer est celle du composant image.

Information

Pour réaliser le dernier point, il faut bien se rappeler de la séance précédente et de la partie sur l'`inspecteur` des objets en `Unity`. En effet, si on regarde bien, nous avons différents composants pour le champs de saisie. Le composant qui nous intéresse est l'**image** permettant de changer l'attribut `color` pour répondre à nos conditions. Pour cela, il nous suffit, à partir du `GameObject`, de récupérer l'objet souhaité :

```
Image image = gameObject.GetComponent<Image>();
```

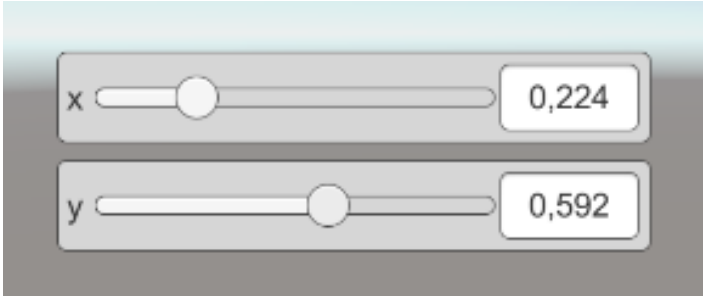
J'insiste que `GetComponent` récupère le premier composant du type souhaité dans le `GameObject` courant, il existe une version permettant de récupérer tous les composants d'un type cible.

```
Image[] images = gameObject.GetComponents<Image>();
```

Réaliser le widget voulu, en attachant une importance à la hiérarchie dans la structure de votre projet et aux noms que vous adoptez (harmonisation et uniformisation).

2 Widget : ComplexSlider

Dans la section précédente, nous avions un unique widget. Ici, nous allons composer un widget complexe à partir de 3 widgets basiques. Pour cela, nous allons réaliser des curseurs complexes en composant un `texte` qui servira de label, d'un `slider` basique et d'un `champ de saisie` sur les nombres pour visualiser la valeur du `slider` et/ou la modifier manuellement. Ci-dessous, vous avez une image qui illustre deux `ComplexSlider` pour choisir des coordonnées `(x,y)`.



Information

Généralement un widget complexe doit avoir un panel à la base pour servir de fond visuel que vous opacifierez par défaut et qui permet de hiérarchiser les 3 sous-widgets le composant.

Vous devez réaliser un tel composant puis réaliser un unique script qui gouverne tous les aspects comportementaux. En particulier, lorsqu'une valeur est modifiée, cela impacte l'autre widget pour avoir toujours une cohérence entre le champ de saisie et le `slider` basique.

Pour cela, nous allons exploiter la hiérarchie de notre widget. Il existe deux mécaniques pour retrouver les enfants :

1. *via* les fonctions de recherche basées sur leur nom,
2. *via* les mécaniques de recherche sur un type souhaité.

Réfléchissez sur les avantages et inconvénients des deux mécaniques en lisant la documentation associée à ces familles de fonctions : `https://docs.unity3d.com/ScriptReference/GameObject.html`. En particulier, que se passe-t-il si plusieurs objets sont du même type ? Ou si l'utilisateur modifie le nom d'un `sous-widget` ?.

3 Exporter son travail

Bravo, vous avez fait le plus gros. Demander à l'enseignant qu'il vérifie votre développement ou critique vos noms et autres petits détails.

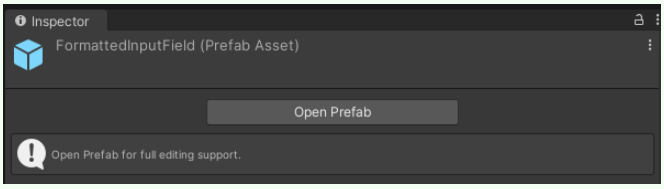
3.1 Préfabriqué

L'intéret de développer un widget est de pouvoir le réutiliser plusieurs fois sans devoir faire plusieurs manipulations identiques ou des `copier-coller`. Pour cela, `Unity` a la possibilité de créer des `Préfabs`, une sorte de sauvegarde de votre réalisation au sein d'un projet !

La procédure est assez simple : lorsque vous avez fini un `widget` qui n'a pas de dépendance extérieure, c'est-à-dire que le `widget` est autonome (sinon les dépendances risquent aussi d'être sauvegardées) : il suffit de glisser le `GameObject` de la hiérarchie à la zone des `Assets`. Ainsi l'icône du `Gameobject` devient bleue ! C'est ainsi qu'on sait qu'il s'agit d'un `Prefab`.

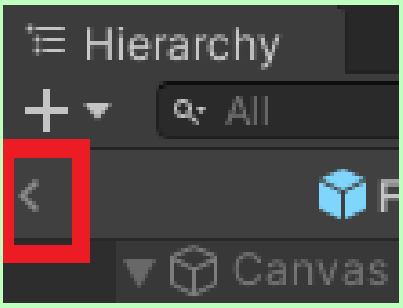
Information

Lorsque vous sélectionnez un `Prefab` dans la zone des `Assets`, vous avez un nouveau bouton dans l'`Inspecteur` qui vous permet de modifier ce `Prefab`.



Attention

Quand vous modifiez un `Prefab`, vous êtes dans un environnement particulier. Il est essentiel de revenir au plus vite dans le mode standard, en particulier pour sauvegarder et ainsi éviter des petits problèmes.
Pour quitter, il faut cliquer sur le bouton "Retour" (noté par $'<'$) au niveau de la hiérarchie.

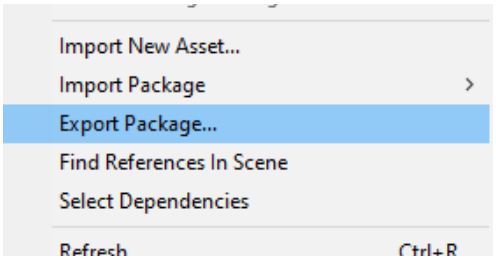


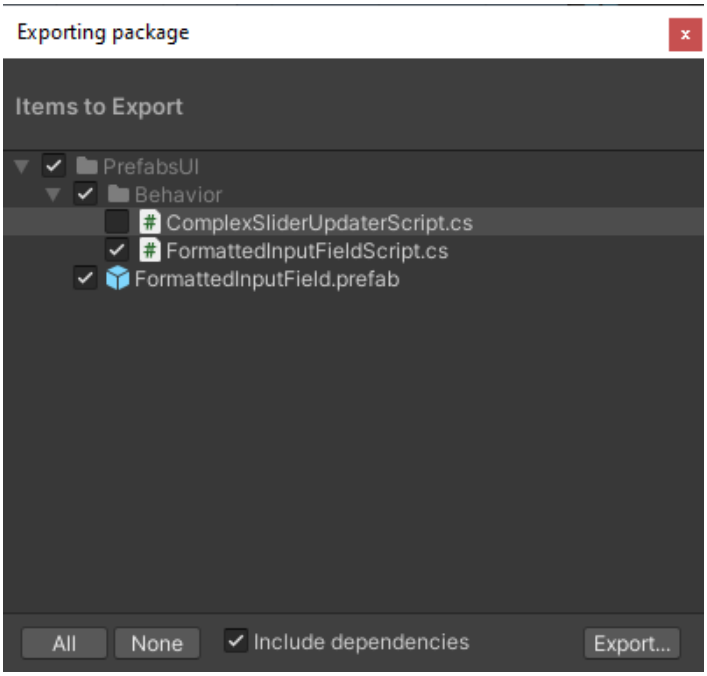
Réaliser des `Prefabs` de vos widgets en les ayant nettoyés si besoin AVANT ! Ensuite, instancier pour expérimenter deux ou trois de vos `Prefabs`. Pour cela, il suffit de glisser votre `Prefab` depuis l'`Assert` vers la vue 3D ou la hiérarchie à l'endroit souhaité.

3.2 Export

Maintenant que vous avez la possibilité de sauvegarder et réutiliser votre production au sein d'un projet, il est fréquent de devoir les exporter pour les utiliser dans d'autres projets. Pour cela, il faut exporter des `Prefabs` dont vous connaissez les dépendances, en sélectionnant votre `Prefab` dans les `Assets` et en faisant un clique droit dessus (attention, on vous demande de tester l'export d'un widget dans un premier temps, puis plusieurs widgets dans un second temps afin que vous compreniez les dépendances). Dans le menu contextuel, sélectionner `'Export Package...'`.

Dans la nouvelle fenêtre, vous devez sélectionner les bonnes dépendances de votre widget. En particulier, `Unity` sélectionne par défaut tous les scripts sans distinction, car il n'arrive pas à calculer les dépendances correctement. Vous devez donc sélectionner les bonnes dépendances de scripts manuellement. Puis, cliquez sur le bouton `Export...` pour sauvegarder le package `Unity`.





3.3 Import

Pour tester nos paquets `Unity` créés à l'étape précédente, c'est très simple ! Il suffit de créer un nouveau projet `Unity` et glisser votre fichier `.unitypackage` dans les `Assets`, puis se laisser guider par le menu.

Information

Une alternative consiste à faire juste un clique droit dans la zone des `Assets` et cliquer sur l'import d'un paquet personnalisé.

Votre widget est prêt à l'emploi. La mécanique des `Unity packages` est l'une des façons de se partager le travail lorsqu'on est plusieurs sur un même projet et que cela s'y prête bien.

4 Toujours plus loin

4.1 Des Prefabs partout !

Reprendre votre TP précédent ou un nouveau projet qui exploite vos nouveaux widgets, histoire de voir une application complète et comprendre que votre `Prefab` doit bien être autonome pour embarquer uniquement son comportement intrinsèque !

4.2 Retour sur la multi-résolution

En reprenant l'hyperlien cité en introduction, vérifier que les *ancres* des éléments du `Canvas` sont bien définies puis jouer avec le composant `Canvas Scaler` pour tester le positionnement de ces éléments.

Tester également les résolutions proposées dans le `GameView`. Enfin, créer une résolution personnalisée (imiter un mode "Portrait" par exemple) et vérifier si les widgets sont toujours positionnés correctement.

4.3 Orientation du Canvas

À partir du lien UICanvas, expérimenter les modes de rendu `"Screen Space - Camera"` et `"World Space"` pour modifier l'orientation et la position du `Canvas` et de ses enfants.